P08- Pruebas de aceptación: Selenium WebDriver

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P08) termina justo ANTES de comenzar la siguiente sesión de prácticas (P09) en el laboratorio (los grupos de los lunes, por lo tanto, el próximo lunes, los de los martes, el próximo martes, ...). Pasado este plazo los profesores no revisarán vuestro trabajo sobre P08 (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

Pruebas de aceptación de aplicaciones Web

El objetivo de esta práctica es automatizar pruebas de aceptación de pruebas emergentes funcionales sobre una aplicación Web. Utilizaremos la librería Selenium WebDriver, desde el navegador Chrome.

Tal y como hemos explicado en clase, usaremos un proyecto Maven que contendrá únicamente nuestros drivers, puesto que no disponemos del código fuente de la aplicación sobre las que haremos las pruebas de aceptación. Al igual que en la sesión anterior, proporcionamos los casos de prueba obtenidos a partir de un escenario de la aplicación a probar. Recuerda que nuestro objetivo concreto es validar la funcionalidad del sistema.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P08-WebDriver** dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2021-Gx-apellido1-apellido2.

ChromeDriver

La librería webdriver interactúa con el navegador a través un driver. Como vamos a ejecutar nuestros tests a través de Chrome necesitamos usar el driver ***chromedriver***.

Vamos a descargarlo desde: https://chromedriver.storage.googleapis.com/index.html

Necesitamos la versión que coincida con la versión del navegador. Si la versión del navegador es, por ejemplo, 89.0.4389.114, nos descargaremos la versión 89.0.4389.23, para linux, es decir, el fichero: ***chromedriver_linux64.zip*** de la carpeta ***89.0.4389.23.***

El fichero descargado lo copiamos en nuestro $HOME, en la carpeta ***chromedriver***, y lo descomprimimos.

A continuación vamos a añadir la ruta del driver en nuestro PATH, para que esté accesible desde cualquier directorio. Para ello, editamos el fichero oculto ***.profile*** que se encuentra en nuestro $HOME, y añadimos la línea (al final del fichero):

```
export PATH=$PATH:/home/ppss/chromedriver
```

Guardamos los cambios, cerramos la sesión, y volvemos a entrar para que los cambios en el fichero ***.profile*** tengan efecto.

El driver necesita usar la librería libconf 2.4. La instalamos con el comando:

```
> sudo apt-get install libgconf-2-4
```

Finalmente podemos probar el driver desde un terminal tecleando el nombre del ejecutable:

```
> chromedriver
```

Ahora ya estamos en disposición de usar el driver desde nuestro código, a partir de una instancia de tipo ChromeDriver:

```
WebDriver driver = new ChromeDriver();                    (opción 1)
```

De forma alternativa, podríamos copiar el driver (fichero chromedriver), por ejemplo en la carpeta src/test/resources/drivers de nuestro proyecto maven. En ese caso, usaríamos las sentencias:

```
System.setProperty("webdriver.chrome.driver",
                   "./src/test/resources/drivers/chromedriver");       (opción 2)
WebDriver driver = new ChromeDriver();
```

Las dos opciones son válidas. Con la opción 2 conseguimos que nuestro código sea más portable, pero puedes usar la opción 1 puesto que hemos actualizado el PATH del sistema a través del fichero *.profile*.

Cookies

Las cookies son datos almacenados en ficheros de texto en el ordenador. Cuando un servidor web envía una página a un navegador, la "conexión" termina, y el servidor "olvida" cualquier cosa sobre el usuario.

Las cookies se inventaron para resolver el problema de "cómo recordar información sobre el usuario", de forma que cuando un usuario visita una página web, se almacena cierta información en una cookie, que se enviará al servidor, de forma que dicho servidor sabrá que la petición proviene del mismo usuario.

Por lo tanto, las cookies constituyen un mecanismo para mantener el estado en una aplicación Web. Es decir, la idea es permitir que una aplicación web tenga la capacidad de interaccionar con un determinado usuario, diferenciándolo del resto. Si no mantenemos el estado, cada vez que un usuario accede a una página web, el servidor no sabrá si se trata del mismo usuario o si cada petición es de un usuario diferente. Como ejemplo: es como cuando dejamos la ropa en la tintorería, y el dependiente nos da un ticket, de forma que cuando vayamos a recogerla, sabrán que hemos ido antes a dejarla, y qué ropa venimos a recoger. Pues bien, ese ticket es similar a una cookie.

Cada cookie se asocia con una serie de propiedades: nombre, valor, dominio, ruta (path), fecha de expiración, y si segura o no.

Cuando validamos por ejemplo una aplicación de venta on-line, necesitaremos automatizar escenarios de prueba como hacer un pedido, ver el carrito de compra, proporcionar los datos de pago, confirmar el pedido, etc. Si no almacenamos las cookies, necesitaremos loguearnos en el sistema cada vez que ejecutemos cualquiera de los escenarios anteriores, lo cual incrementará el tiempo de ejecución de nuestros tests.

Una posible solución es almacenar las cookies en un fichero, y posteriormente recuperarlas y guardarlas en el navegador. De esta forma podremos "saltarnos" el proceso de login en nuestro test ya que el navegador ya tendrá esta información.

En la carpeta **Plantillas-P08**, hemos proporcionado una clase ***Cookies***, con 3 métodos, para poder:

- almacenar en un fichero de texto en el directorio target, las cookies generadas cuando los logueamos en el sistema,
- leer la información sobre las cookies almacenada en el fichero y guardarla en el navegador
- imprimir por pantalla las cookies almacenadas en nuestro navegador.

Usaremos esta clase en el tercer ejercicio de esta práctica.

Ejercicios

Vamos a implementar nuestros tests de pruebas de aceptación para una aplicación Web denominada Guru Bank, a la que accederemos desde Chrome usando la url http://demo.guru99.com/V5.

Una vez en la página principal, debéis **crearos una cuenta** de administrador para usar la aplicación (siguiendo las instrucciones que se muestran en la parte inferior de dicha página). Recuerda que, tal y como se indica, el usuario y password asignados tienen una validez de 20 días, por lo que si necesitas usar la aplicación pasado ese tiempo deberás crear otra cuenta (y actualizar dicha información en el código de tus tests).

Para los ejercicios de esta sesión crea, en la carpeta *ppss-2021-Gx-apellido1-apellido2/P08-WebDriver/* un proyecto maven, con groupId = **ppss**, y artifactID= **guru99Bank**.

Lo PRIMERO que tienes que hacer es configurar correctamente el pom. Debes incluir las propiedades, dependencias y plugins necesarios para implementar tests unitarios con webdriver. Recuerda que para Maven serán tests unitarios (y se ejecutarán a través del plugin surefire), pero realmente son tests de aceptación, tal y como ya hemos explicado en clase.

Si estuviésemos haciendo las pruebas de aceptación sobre código desarrollado por nosotros, en src/main tendríamos el código fuente a probar, y en src/test tendríamos tanto los tests unitarios (ejecutados con surefire), como el resto de tests (ejecutados con failsafe).

Observaciones sobre esperas implícitas y/o explícitas

Para sincronizar la carga de las páginas con la ejecución de nuestros drivers, usaremos un **wait implícito**. Recuerda que en este caso establecemos el mismo temporizador para todos los *webelements* de las páginas. Sólo se usa una vez, antes de ejecutar cada test. Este código webdriver lo incluiremos en nuestro test (aunque usemos el patrón Page Object) ya que no se verá afectado por posibles cambios de las páginas html a las que accede dicho test. El valor del temporizador será en segundos, y dependerá de la máquina en la que estéis ejecutando los tests. Podéis probar inicialmente con 5 segundos y ajustarlo a un valor mayor o menor si es necesario.

También puedes usar un **wait explícito**, en cuyo caso sólo afectará al elemento al que asociemos el temporizador.

Ejemplo de *wait* **implícito**:

```
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```

Ejemplo de *wait* **explícito** para una ventana de alerta:

```
WebDriverWait wait = new WebDriverWait(driver, 10);
wait.until(ExpectedConditions.alertIsPresent());
```

NOTA: Recuerda que aconsejamos, para realizar este tipo de pruebas, que intentes tener el menor número de aplicaciones abiertas de forma innecesaria, ya que esto puede "ralentizar" la carga de las páginas en el navegador, dificultando así el proceso de testing (generando errores debido a que no se cargan las páginas con la suficiente rapidez).

➪ ➪ Ejercicio 1:Tests Login

Vamos a crear el paquete ***ejercicio1.sinPageObject***, en el que implementaremos dos casos de prueba sin usar el patrón Page Object.

Los casos de prueba a implementar son los siguientes (en una clase **TestLogin**):

- Un primer caso de prueba (con el nombre ***test_Login_Correct()***) en el que accederemos a la aplicación bancaria con un nombre de usuario y contraseña correctos y en pantalla aparecerá el correspondiente mensaje de bienvenida (podéis usar el mensaje completo o parte de él para verificar el resultado).
- Un segundo caso de prueba, al que llamaremos ***test_Login_Incorrect()*** en el que accederemos a la aplicación bancaria con un nombre de usuario y contraseña incorrectos, y debemos verificar que se muestra el mensaje "User is not valid" en un *alert Box.*

  Para implementar el driver necesitas usar la clase Alert, que representa un elemento *Alert Box*..

  Alert Box

  Un *Alert Box* no es más que un pequeño “recuadro” que aparece en la pantalla que proporciona algún tipo de información o aviso sobre alguna operación que intentamos realizar, pudiendo solicitarnos algún tipo de permiso para realizar dicha operación

  A continuación mostramos un ejemplo de algunos métodos que podemos usar:

```
//Operaciones sobre ventanas de alerta
//cambiamos el foco a la ventana de alerta
Alert alert = driver.switchTo().alert();
//podemos mostrar el mensaje de la ventana
String mensaje = alert.getText();
//podemos pulsar sobre el botón OK (si lo hubiese)
alert.accept();
//podemos pulsar sobre el botón Cancel (si lo hubiese)
alert.dismiss();
//podemos teclear algún texto (si procede)
alert.sendKeys("user");
```

  También puedes consultar aquí un ejemplo de uso de *Alert Box* con WebDriver:
  https://www.guru99.com/alert-popup-handling-selenium.html

Crea un elemento de configuración Maven desde IntelliJ (ponle como nombre: ***Test Login***) para invocar al comando:

```
mvn test -Dtest=ejercicio1.sinPageObject.TestLogin
```

Simplemente tienes que rellenar el campo "Command Line" omitiendo "mvn". Recuerda también que, para no "perder" la configuración al subir nuestro proyecto a Bitbucket, tendrás que marcar "Store as Project File" (en la parte superior derecha de la ventana), y guardarlo, por ejemplo, en la carpeta ***src/test/resources/configurations/***

Nota: Si al ejecutar la *configuration* desde IntelliJ te aparece una ventana de alerta con el mensaje:

'Login' is not allowed to run in parallel. Would you like to stop the running one?

Simplemente ignora el mensaje seleccionando "*Cancel*".

➪ ➪ Ejercicio 2:Tests Login usando Page Objects

Vamos a crear el paquete ***ejercicio2.conPO***, en el que implementaremos los dos casos de prueba del ejercicio anterior usando el patrón Page Object. No usaremos la clase *PageFactory*. Recuerda que, además de los drivers, tienes que implementar la/s Page Object de las que dependen los tests, y que éstas estarán en src/main, tal y como hemos explicado en clase.

Los casos de prueba se implementarán en una clase ***TestLogin***, y los nombres de cada caso de prueba serán los mismos que en el ejercicio anterior: *test_Login_Correct()* y *test_Login_Incorrect()*.

En este caso necesitaréis implementar dos *page objects*: *LoginPage* y *ManagerPage.*

Para ejecutar los tests crea un elemento de configuración Maven desde IntelliJ (ponle como nombre = ***Test LoginPO***) con el comando:

```
mvn test –Dtest=ejercicio2.conPO.TestLogin
```

Recuerda marcar "*Store as Project File*" para guardar la configuración en la *carpeta test/resources/configurations*.

➪ ➪ Ejercicio 3: Tests NewClient

Vamos a crear el paquete ***ejercicio3.conPOyPFact***, en el que implementaremos dos nuevos casos de prueba usando el patrón *Page Object*, junto con la clase *PageFactory*.

Necesitarás implementar las Page Objects necesarias en src/main, pero en el mismo paquete que nuestros drivers, en este caso serán: *ManagerPage* y *NewCustomerPage*

No vamos a necesitar la clase LoginPage ya que vamos a ejecutar los tests de forma que nos "saltemos" el paso de loguearnos en el sistema. Para ello usaremos el método *Cookies.storeCookiesToFile(login, password)* invocándolo UNA sóla vez y antes de ejecutar cualquier test.

Posteriormente y antes de ejecutar cada test, tendremos que guardar las cookies en el navegador invocando al método *Cookies.loadCookiesFromFile(driver)*. Esto lo haremos antes de acceder a la página inicial, que en este caso, no será la página de login, sino que directamente será la página inicial del Manager (http://demo.guru99.com/V5/manager/Managerhomepage.php)

La clase *Cookies* estará en src/main junto con las *Page Objects*.

Los casos de prueba que vamos a implementar son los siguientes (en la clase ***TestNewClientCookies***):

- ***testTestNewClientOk()***: después de haber accedido al banco con nuestras credenciales (este paso no será necesario ya que estaremos identificados por el sistema gracias a las cookies que hemos guardado en el el fichero), daremos de alta a un nuevo cliente, comprobaremos que el proceso termina correctamente (gracias a un mensaje en la pantalla del navegador), y finalmente regresaremos a la pantalla inicial del Manager (lo cual tendremos también que comprobar).

  Como regla general, SIEMPRE que cambiemos de pantalla en el navegador, debemos incluir el *Asssert* correspondiente para asegurarnos de que estamos en la página correcta.

IMPORTANTE!!

Cuando creamos un nuevo cliente, la aplicación le asigna de forma automática un identificador único. Dicho valor será necesario para realizar cualquier operación con el cliente (por ejemplo editar sus datos, borrarlo...). Por lo tanto, deberás almacenar dicho valor (como un atributo en la clase que contiene los tests) para poder consultarlo desde todos los tests que necesiten trabajar con dicho cliente. También puedes mostrar dicho valor por pantalla durante la ejecución del test.

Para recordar los datos del nuevo cliente, vamos a realizar una captura de pantalla durante la ejecución del test, una vez que enviemos los datos del nuevo cliente al servidor a través del botón del formulario correspondiente.

Para generar una captura de pantalla y guardarla en un fichero (puedes usar el nombre que consideres oportuno) podemos hacerlo con:

```
File scrFile = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
File destFile = new File("./target/"+filename+".png");
Files.copy(scrFile.toPath(),destFile.toPath(),StandardCopyOption.REPLACE_EXISTING);
```

Para poder reutilizar estas líneas si fuese necesario hacer más capturas de pantalla, hemos proporcionado (en la carpeta de plantillas) el método ScreenShot.captura(driver,filename) con dicho código.

La clase ScreenShot deberá estar en src/main junto con las *Page Objects*

Los datos concretos para el nuevo cliente, pueden ser éstos:

- Nombre del cliente: tu login del correo electrónico de la ua
- Género: "m" (o "f")
- Fecha de nacimiento: tu fecha de nacimiento en formato "aaaa-mm-dd" (*Ver nota)
- Dirección: "Calle x"
- Ciudad: "Alicante", (para el campo State puedes poner el mismo)
- Pin: "123456" (puedes poner cualquier otra secuencia de 6 dígitos)
- Número de móvil: "99999999"
- e-mail: tu email institucional
- password: "123456", (o cualquier otra secuencia de dígitos)

Una vez introducidos los datos pulsaremos sobre el botón "Submit" (asegúrate de que haces scroll de la pantalla para que el botón submit esté visible, lo explicamos a contiuación).

El resultado del test será "pass" si hemos conseguido crear el nuevo cliente. Para ello podemos comprobar que en la nueva página aparece el texto (o una parte de él) "Customer Registered Successfully!!!".

> **NOTA**: Para introducir la fecha usaremos tres veces el método *sendKeys* con el día, el mes y el año, respectivamente. Ten en cuenta que el formato para el dia sera "dd", para el mes "mm" y para el año "yyyy". Por ejemplo, "06", "05", "2003"

Cómo realizar "scroll" de la ventana del navegador

Cuando vamos a introducir los datos del nuevo cliente que queremos crear, observa que no "caben" en la pantalla todos los cuadros de texto, y que para introducir todos los datos, necesitaremos hacer "scroll" en la ventana del navegador (de no hacerlo así obtendremos un error, puesto que no podemos interaccionar con elementos de la página que no están visibles).

Para ello necesitaremos usar el método javascript ***scrollIntoView()***. A continuación mostramos un ejemplo en el que hacemos scroll del contenido de la ventana hasta que sea visible el webelement al que hemos denominado "submit" y que representa el botón para enviar el formulario con los datos del cliente.

```
...
//Si no hacemos esto, el botón submit no está a la vista y
//no podremos enviar los datos del formulario
JavascriptExecutor js = (JavascriptExecutor) driver;
//This will scroll the page till the element is found
js.executeScript("arguments[0].scrollIntoView();", submit);
//ahora ya está visible el botón en la página y ya podemos hacer click sobre él
submit.click();
...
```

Puedes consultar aquí un ejemplo de uso de *Scroll* con WebDriver:
https://www.guru99.com/scroll-up-down-selenium-webdriver.html

Otras observaciones a tener en cuenta:

- Recuerda que si usamos el patrón Page Object nuestros tests NO deben contener código Webdriver.
- Recuerda que debes verificar, cada vez que cambiemos de página, que estamos en la página correcta, para ello puedes utilizar el título de la página o alguna información de la misma, por ejemplo el login del administrador (en el caso de la página resultante de hacer login).

- El campo e-mail del cliente es único para cualquier cliente. Si intentamos dar de alta un cliente con un e-mail ya registrado, la aplicación nos mostrará un mensaje indicando que no se puede repetir el valor de dicho campo.
- Adicionalmente, para poder repetir la ejecución del test debemos "eliminar" el cliente que acabamos de crear después de ejecutar con éxito el test. Para ello tendrás que crear una nueva *page object*, que puedes llamar "DeleteCustomerPage" y que representa la página de nuestra aplicación para borrar un cliente. Una vez implementada la nueva page object, modifica el test para borrar el nuevo cliente creado (tendrás que utilizar el identificador del cliente para poder borrarlo), y además aceptar los mensajes de dos alertas que nos aparecerán para confirmar que queremos borrar dicha información.
- El borrado del cliente puedes hacerlo en un método privado que invocarás desde el método anotado con @AfterEach.

- ***testTestNewClientDuplicate()***: en este segundo test, se trata de crear un nuevo cliente usando un e-mail que ya existe. En este caso, después de introducir los datos del cliente y pulsar el botón "Submit" nos debe aparecer un mensaje de alerta con el mensaje "Email Address Already Exist !!". Para garantizar que el cliente que introducimos como entrada ya existe, debes añadirlo previamente, y a continuación, repetir la operación para asegurarnos de que efectivamente se trata de un cliente repetido. Debemos comprobar que la aplicación nos muestra el mensaje anterior y que no nos deja insertar dos veces el mismo cliente. Al igual que antes, debes borrar los datos del cliente creado después de ejecutar el test para poder repetir su ejecución con las mismas condiciones.

Puedes crear un elemento de configuración Maven desde IntelliJ (nombre = ***TestNewClientCookies***) con el comando:

```
mvn test -Dtest=TestNewClientCookies
```

Recuerda marcar "*Store as Project File*" para guardar la configuración en la *carpeta test/resources/configurations*.

Modo "*headless*"

Podemos ejecutar los tests sin necesidad de mostrar el navegador, con lo cual ahorraremos tiempo de ejecución. Para ello tenemos que configurar el driver en modo *headless*. La idea es cambiar el modo de ejecución de los tests desde el proceso de construcción. Para ello tendremos que:

- Modificar la configuración del plugin ***surefire***. Vamos a añadir una "propiedad" en la etiqueta <systemPropertyVariables>. Llamaremos a esta propiedad "***chromeHeadless***" (es una elección personal, podemos elegir cualquier nombre que consideremos oportuno). El valor de la propiedad "*chromeHeadless*" tendrá como valor el de la *<property>* ***headless.value*** (de nuevo este nombre depende de nuestra elección personal).

```
<properties>
    ...
    <headless.value>false</headless.value>
</properties>

<plugin>
    ...
    <artifactId>maven-surefire-plugin</artifactId>
    ...
    <configuration>
        <systemPropertyVariables>
            <chromeHeadless>${headless.value}</chromeHeadless>
        </systemPropertyVariables>
    </configuration>
</plugin>
```

- En el código del test (en el método anotado con *@BeforeEach*) creamos la instancia de *ChromeDriver* activando el modo *headless.* Para ello usaremos el valor de la propiedad *chromeHeadless* que hemos definido en la configuración del plugin *surefire*.

```
ChromeOptions chromeOptions = new ChromeOptions();
//recuperamos el valor de la propiedad chromeHeadless definida en surefire
boolean headless = Boolean.parseBoolean(System.getProperty("chromeHeadless"));
//el método setHeadless() cambia la configuración de Chrome a modo headless
chromeOptions.setHeadless(headless);
//ahora creamos una instancia de CromeDriver a partir de chromeOptions
driver = new ChromeDriver(chromeOptions);
```

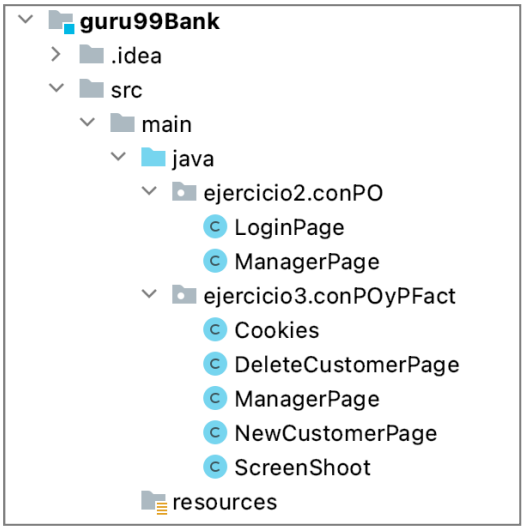
- Ahora podemos activar el modo headless usando el valor "true" para la *property headless.value* de nuestro *pom*.

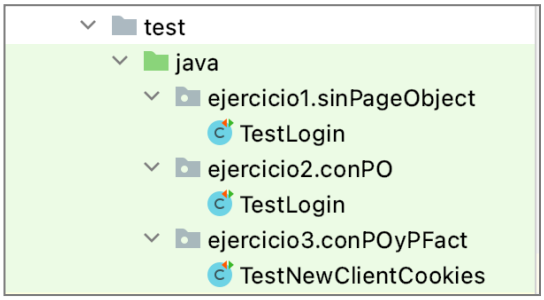
```
mvn test -Dtest=TestNewClientCookies -Dheadless.value=true
```

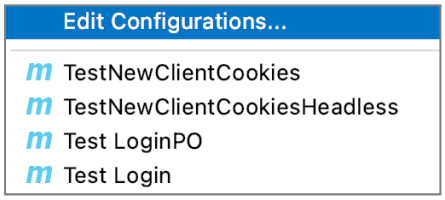
Crea un elemento de configuración Maven desde IntelliJ (nombre = ***TestNewClientCookiesHeadless***) con el comando anterior y pruébalo. Observarás que las capturas de pantalla se realizan con independencia de si estamos ejecutando nuestros tests en modo *headless* o no.

Recuerda marcar "*Store as Project File*" para guardar la configuración en la carpeta *test/resources/configurations*.

Finalmente, mostramos capturas de pantalla del proyecto maven para esta práctica, así como el conjunto de "*configurations*" que debéis crear.:







Resumen

¿Qué conceptos y cuestiones me deben quedar CLAROS después de hacer la práctica?

AUTOMATIZACIÓN DE PRUEBAS DE ACEPTACIÓN

- Consideraremos el caso de que nuestro SUT sea una aplicación web, por lo que usaremos webdriver para implementar y ejecutar los casos de prueba. La aplicación estará desplegada en un servidor web o un servidor de aplicaciones, y nuestros tests accederán a nuestro SUT a través de webdriver, que será nuestro intermediario con el navegador (en este caso usaremos Chrome, junt con el driver correspondiente)
- Si usamos webdriver directamente en nuestros tests, éstos dependerán del código html de las páginas web de la aplicación a probar, y por lo tanto serán muy "sensibles" a cualquier cambio en el código html. Una forma de independizarlos de la interfaz web es usar el patrón PAGE OBJECT„ de forma que nuestros tests NO contendrán código webdriver, independizándolos del código html. El código webdriver estará en las page objects que son las clases que dependen directamente del código html, a su vez nuestros tests dependerán de las page objects.
- Junto con el patrón Page Object, podemos usar la clase PageFactory para crear e inicializar los atributos de una Page Object. Los valores de atributos se inyectan en el test mediante la anotación @Findby, y a través del localizador correspondiente. Esta inyección se realiza de forma "lazy", es decir, los valores se inyectan justo antes de ser usados.
- Con webdriver podemos manejar las alertas generadas por la aplicación a probar, introducir esperas (implícitas y explícitas), realizar scroll en la pantalla del navegador, agrupar elementos, capturar la pantalla del navegador y manejar cookies, entre otras cosas.
- El manejo de las cookies del navegador nos será útil para acortar la duración de los tests, ya que podremos evitar loguearnos en la aplicación para probar determinados escenarios.
- También podremos acortar los tiempos de ejecución de nuestros tests si los ejecutamos en modo headless. Podemos decidir si vamos a ejecutar o no nuestros tests en modo headless aprovechando la capacidad del plugin surefire y/o failsafe, de hacer llegar a nuestros tests ciertas propiedades definidas por el usuario. De esta forma podremos, cuando lancemos el proceso de construcción, ejecutar en ambos modos sin modificar el código.